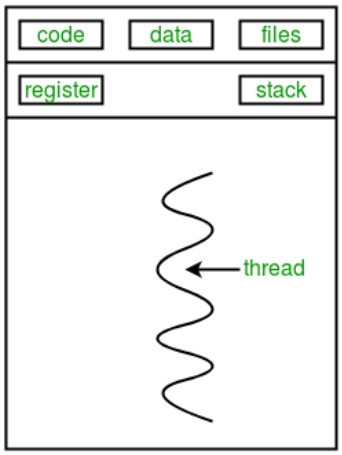
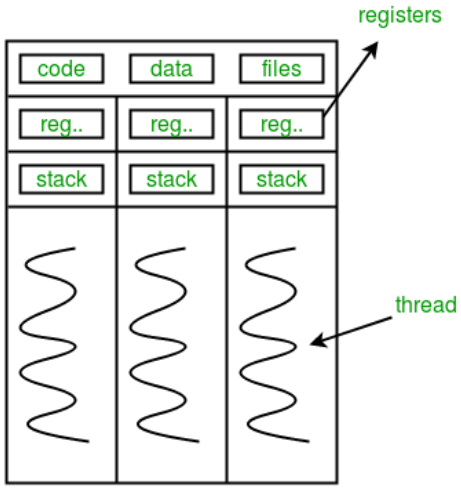
UD02: Programación multihilo



single-threaded process



multithreaded process

1. Fundamentos dela programación multihilo

En muchas ocasiones las sentencias que forman un programa deben ejecutarse de manera secuencial, ya que, entre todas, forman una lista ordenada de pasos a seguir para resolver un problema. Otras veces, dichos pasos no tienen por qué ser secuenciales, pudiéndose realizar varios a la vez.

Otras veces, disponer de simultaneidad en la ejecución no es opcional, sino necesario, como ocurre en las aplicaciones web. Si no se atendiesen simultáneamente las peticiones que recibe un servidor web, cada usuario debería esperar hasta que todos los usuarios que llegaron antes que él fuesen atendidos para obtener las páginas o recursos solicitados.

En ambos casos, la solución se encuentra en la programación multihilo, una técnica utilizada para conseguir procesamiento simultáneo.

Aunque plena de posibilidades, esta técnica no está exenta de condiciones y restricciones. En esta unidad se presentan las herramientas de procesamiento multihilo disponibles en el lenguaje Java, junto con los aspectos teóricos referentes al análisis de los procesos para determinar si es posible ejecutarlos con múltiples hilos, así como a detectar y prevenir los conflictos que surgen como consecuencia de la programación concurrente.

Desde el punto de vista del sistema operativo, un programa informático en ejecución es un proceso que compite con los demás procesos por acceder a los recursos del sistema. Visto desde dentro, un programa es, básicamente, una sucesión de sentencias que se ejecutan una detrás de otra.

El sistema operativo se encarga de hacer convivir los diferentes procesos y de repartir los recursos entre sí, por lo que cuando se programa no hay que tener en cuenta aspectos relacionados con cómo se van a gestionar los procesos o cómo estos van a tener acceso a los recursos. Salvo el optimizar el uso de memoria y conseguir que los algoritmos sean eficientes, los programadores no tienen que preocuparse del resto: el sistema operativo se encarga de la concurrencia a nivel de proceso.

Es dentro del interior de los procesos donde la programación tiene algo que decir con respecto a la concurrencia, mediante la programación multihilo.

> Un hilo, también conocido como thread, es una pequeña unidad de computación que se ejecuta dentro del contexto de un proceso. Todos los programas utilizan hilos.

En el caso de un programa absolutamente secuencial, el hilo de ejecución es único, lo que provoca que cada sentencia tenga que esperar que la sentencia inmediatamente anterior se ejecute completamente antes de comenzar su ejecución. Esto no quiere decir en ningún caso que no existan estructuras de control, como if o while, sino que las sentencias se van ejecutando una detrás de otra y no de manera simultánea.

En cambio, en un programa multihilo, algunas de las sentencias se ejecutan simultáneamente, ya que los hilos creados y activos en un momento dado acceden a los recursos de procesamiento sin necesitar esperar a que otras partes del programa terminen.

A continuación, se citan algunas de las características que tienen los hilos:

- Dependencia del proceso. No se pueden ejecutar independientemente. Siempre se tienen que ejecutar dentro del contexto de un proceso.

- A Ligereza. Al ejecutarse dentro del contexto de un proceso, no requiere generar procesos nuevos, por lo que son óptimos desde el punto de vista del uso de recursos. Se pueden generar gran cantidad de hilos sin que provoquen pérdidas de memoria.
- Compartición de recursos. Dentro del mismo proceso, los hilos comparten espacio de memoria. Esto implica que pueden sufrir colisiones en los accesos a las variables provocando errores de concurrencia.
- Paralelismo. Aprovechan los núcleos del procesador generando un paralelismo real, siempre dentro de las capacidades del procesador.
- Concurrencia. Permiten atender de manera concurrente múltiples peticiones. Esto es especialmente importante en servidores web y de bases de datos, por ejemplo.

Para ilustrar de manera más precisa cómo se comporta un programa de un único hilo frente a los programas de múltiples hilos se puede establecer la siguiente analogía.

El camarero de una cafetería recibe a un cliente que le pide un café, una tostada y una tortilla francesa. Si el camarero trabaja como un proceso de un único hilo pondrá la cafetera a preparar el café, esperará a que termine para poner el pan a tostar y no pedirá la tortilla a la cocina hasta que el pan no esté tostado. Probablemente, cuando todo esté dispuesto para servir, el café y la tostada estarán fríos y el cliente aburrido de esperar.

Los camareros (la gran mayoría de ellos) suelen trabajar como procesos multihilo: ponen la cafetera a preparar el café, el pan a tostar y piden a la cocina los platos sin esperar a que cada una de las demás tareas esté terminada. Dado que cada una de ellas consume diferentes recursos, no es necesario hacerlo. Transcurrido el tiempo que tarda en estar lista la tarea más lenta, el cliente estará atendido.

Siguiendo con la analogía, al igual que pasa en los ordenadores, los recursos de una cafetería son limitados, por lo que hay una restricción física que impide que se hagan más tareas simultáneamente de las que permiten los recursos. Si solo hay una tostadora con capacidad para dos rebanadas de pan no se podrán tostar más de dicho número en un mismo instante de tiempo.

La programación que permite realizar este tipo de sistemas se llama multihilo, concurrente o asíncrona.

1.1. Programación secuencial o de Único hilo

El siguiente ejemplo programado en Java ilustra cómo se comporta un programa que se ejecuta en un único hilo, así como las consecuencias que esto implica. El programa está compuesto por una única clase (representa un ratón) compuesta por dos atributos: el nombre y el tiempo en segundos que tarda en comer. En el método main se instancian varios objetos (ratones) y se invoca al método comer de cada uno de ellos. Este método muestra un texto por pantalla cuando comienza, realiza una pausa de la duración en se- gundos (con el método sleep de la clase Thread) que indica el parámetro tiempoAlimentacion y, finalmente, muestra otro texto por pantalla cuando finaliza.

Consulta el Ejemplo01

1.2. Programación concurrente o multihilo

El ejemplo anterior se puede programar de manera concurrente de forma sencilla, ya que no hay ningún recurso compartido. Para ello, basta con convertir cada objeto de la clase Raton en un hilo (thread) y programar aquello que se quiere que ocurra concurrentemente dentro del método run. Una vez creadas las instancias, se invoca al método start de cada una de ellas, lo que provoca la ejecución del contenido del método run en un hilo independiente.

Consulta el Ejemplo02

En Java existen dos formas para la creación de hilos:

- Implementando la interface java.lang.Runnable.
- Heredando de la clase java.lang.Thread.

La implementación de la interface Runnable obliga a programar el método sin argumentos run.

```
1  public class HiloViaInterface implements Runnable {
2      @override
3      public void run() {
4      }
5  }
```

Heredando de la clase Thread esto no es obligatorio porque dicha clase ya es una implementación de la interface Runnable.

```
1  public class HiloViaHerencia extends Thread {
2  }
```

No obstante, crear un hilo heredando de Thread «necesita» la implementación del método sin argumentos `run`, ya que de lo contrario la clase no tendrá un comportamiento multihilo.

Consulta el Ejemplo3

> Es habitual que, en las primeras tomas de contacto con los hilos en Java, los programadores intenten ejecutar el método run en lugar de ejecutar el método start. Desde el punto de vista de la compilación no habrá problema: el programa se compilará sin errores.
>
> Desde la perspectiva de la programación concurrente el resultado no será el adecuado, ya que los hilos se ejecutarán uno detrás de otro, de manera secuencial, no obteniendo ninguna mejora frente a la programación secuencial convencional.

La creación de threads mediante la implementación de la interface Runnable tiene una ventaja clara sobre la herencia de la clase Thread: al ser Java un lenguaje que no admite herencia múltiple, heredar de dicha clase impide otro tipo de herencias, limitando las posibilidades de diseño del software.

Por otra parte, a través de la implementación mediante la interface Runnable se pueden lanzar muchos hilos de ejecución sobre un único objeto, frente a la herencia de Thread que creará un objeto por cada hilo.

En el siguiente ejemplo se implementa un hilo mediante la interface `Runnable` para crear múltiples hilos a partir de un único objeto. En el hilo, un atributo de instancia llamado `alimentoConsumido` se incrementa en 1 durante la ejecución del método `comer`, invocado en el método `run`. Se puede observar en el método `main` cómo se crea una única instancia de la clase

`RatonSimple` y se crean cuatro hilos que la ejecutan.

Consulta el Ejemplo04 y Ejemplo04bis

> Debido a la naturaleza de la programación multihilo, las salidas de las ejecuciones de los programas de ejemplo pueden no coincidir con las presentadas en el libro. De hecho, en distintas ejecuciones en la misma máquina los resultados pueden variar.

1.3. Estados de los hilos

Durante el ciclo de vida de los hilos, estos pasan por diversos estados. En Java, están recogidos dentro de la enumeración State contenida dentro de la clase java.lang.Thread.

El estado de un hilo se obtiene mediante el método getState() de la clase Thread, que devolverá algunos de los valores posibles recogidos en la enumeración indicada anteriormente.

Los estados de un thread son los que se muestran en la siguiente tabla:

Estado	Valor en `Thread.State`	Descripción
Nuevo	`NEW`	El hilo está creado, pero aún no se ha arrancado.
Ejecutado	`RUNNABLE`	El hilo está arrancado y podría estar en ejecución o pendiente de ejecución.
Bloqueado	`BLOCKED`	Bloqueado por un monitor.
Esperando	`WAITING`	El hilo está esperando a que otro hilo realice una acción determinada.
Esperando un tiempo	`TIME_WAITING`	El hilo está esperando a que otro hilo realice una acción determinada en un período de tiempo concreto.
Finalizado	`TERMINATED`	El hilo ha terminado su ejecución.

En el siguiente código de ejemplo, se almacenan en un `ArrayList` los estados por los que pasa un hilo que contiene en su interior una llamada al método `sleep`. Se utiliza la clase `RatonSimple` que implementa `Runnable` de los ejemplos anteriores.

Consulta el Ejemplo05

> Todos los hilos pasan por los estados **NEW**, **RUNNABLE** y **TERMINATED**. El resto de los estados están condicionados a circunstancias propias de la ejecución.

1.4. Introducción a los problemas deconcurrencia

En programación concurrente las dificultades aparecen cuando los distintos hilos acceden a un recurso compartido y limitado. Si en el ejemplo de los ratones (el físico, no el programático), estos comiesen a través de un dispositivo al que solo pudiesen acceder de uno en uno, la concurrencia sería imposible. En el ejemplo programático el problema es el mismo, pero no existe la restricción física, por lo que nada impide que los hilos accedan al recurso compartido y es en ese momento donde aparecerán los problemas.

El acceso a los recursos compartidos limitados, por lo tanto, se debe gestionar correctamente. Por suerte, existen técnicas, clases y librerías que implementan soluciones, por lo que únicamente hay que conocerlas y saberlas utilizar en el momento adecuado.

Un aspecto importante referente a los problemas de concurrencia es que no siempre provocan un error en tiempo de ejecución. Esto significa que en sucesivas ejecuciones del programa multihilo el error se producirá en algunas de ellas y en otras no. Frente al determinismo de los programas secuenciales (los mismos datos de entrada generan siempre las mismas salidas), en el entorno multihilo los factores que determinan las condiciones de ejecución pueden tener su origen en elementos sobre los que el programador no tiene capacidad de influir, como el sistema operativo.

La dificultad, en este tipo de programación, reside en el hecho de que el programador tiene que saber de forma anticipada que el error se puede producir en una sentencia o bloque de sentencias, por cómo esta se ejecuta y qué recursos utiliza. No basta simplemente con realizar pruebas convencionales para determinar que el algoritmo está bien programado. Además, hay que «saber» que está bien programado.

Como se ha indicado anteriormente existen soluciones para todos los problemas, pero en esta afirmación hay una verdad a medias. Los problemas de la programación concurrente se resuelven, en ciertos casos, convirtiendo en secuenciales determinadas partes del código. Si no se programa correctamente, se corre el riesgo de convertir todo el programa en secuencial lo que evitaría, efectivamente, los problemas de concurrencia ya que esta habría dejado de existir.

2. Programación multihilo: clases y librerías

2.1. Paquete `java.lang`

Java dispone de un importante número de clases destinadas a la programación multihilo. Estas clases tienen múltiples aplicaciones, desde la propia construcción de los hilos hasta dar soporte a estructuras de datos consistentes cuando varios hilos acceden a ellas.

Estas clases se encuentran agrupadas en dos paquetes principales: el paquete `java.lang` y el paquete `java.util.concurrent`.

El paquete `java.lang` es un paquete importado por defecto en Java (contiene las clases básicas, incluida `Object` como clase raíz de la jerarquía de clases del lenguaje) por lo que no hay que importar ninguna librería para utilizar sus clases.

Dentro de este paquete se encuentran la interfaz `Runnable` y la clase `Thread`, como elementos fundamentales para la construcción de hilos. También se encuentran las clases `Timer` y `TimerTask`

Nombre	Tipo	Descripción
`Runnable`	Interface	Esta interface debe implementarse por aquellas clases que se quieran ejecutar como un `thread`. Define un método sin argumentos `run`.
`Thread`	Clase	Esta clase implementa la interface `Runnable`, siendo un hilo en sí misma. Una clase que hereda de `Thread` y que sobrescribe el método `run` se ejecuta de manera concurrente si se invoca al método `start`. Con la clase `Thread` se puede envolver un objeto que implemente `Runnable` provocando que se ejecute en un hilo independiente.
`Timer`	Clase	Permite la programación de la ejecución de tareas en diferido y de forma repetitiva mediante el método `schedule`. Este método espera tareas programables, representadas por objetos de la clase `TimerTask`, así como información del momento de inicio de la tarea y del tiempo que debe transcurrir entre cada una de las ejecuciones de la misma.
`TimerTask`	Clase abstracta	Heredando de esta clase y sobrescribiendo el método `run` se puede crear una tarea programable con la clase `Timer`.

Consulta el Ejemplo06

2.2. Paquete `java.util.concurrent`

En este paquete se incluyen un interesante conjunto de clases y herramientas relacionadas con la programación concurrente. A continuación, se muestran algunos de los elementos más relevantes del mismo.

2.2.1. Executor

Es una interface para la definición de sistemas multihilo. Permite ejecutar tareas de tipo `Runnable`. Algunas de sus interfaces derivadas, así como clases directamente relaciona das se muestran en la siguiente tabla.

Nombre	Tipo	Descripción
`ExecutorService`	Interface	Subinterface de Executor, permite gestionar tareas asíncronas
`SchedulledExecutorService`	Interface	Permite la planificación de la ejecución de tareas asíncronas.
`Executors`	Clase	Fábrica de objetos `Executor`, `ExecutorServices`, `ThreadFactory` y `Callable`.
`TimeUnit`	Enumeración	Proporcionarepresentaciones de unidades de tiempo con distinta granularidad, desde días hasta nanosegundos.

Consulta el Ejemplo07

2.3. Queues

Java proporciona componentes que resuelven todo el espectro de necesidades relacionado con las colas para crear entornos seguros de ejecución de soluciones de mensajería, gestión de colas de trabajo y sistemas basados en esquemas productor-consumidor en entornos concurrentes.

Las clases más relevantes de este grupo se recogen en la siguiente tabla.

Nombre	Tipo	Descripción
`ConcurrentLinkedQueue`	Clase	Una cola enlazada resistente a hilos.
`ConcurrentLinkedDeque`	Clase	Una cola doblemente enlazada resistente a hilos.
`BlockingQueue`	Interface	Una cola que incluye bloqueos de espera para la gestión del espacio. Algunas de sus implementaciones son `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue` y `DelayQueue`.
`TransferQueue`	Interface	Un tipo de `BlockingQueue` especialmente diseñado para mensajería.
`BlockingDeque`	Interface	Una cola doblemente enlazada que incluye bloqueos de espera para la gestión del espacio. Dispone de una implementación en la clase `LinkedBlockingDeque`.

Consulta el Ejemplo08 y Ejemplo08bis

2.4. Sincronizadores

El paquete `java.util.concurrent` proporciona cinco clases específicas para conseguir que la concurrencia entre hilos se desarrolle de manera correcta. Estas clases son las que se recogen en la siguiente tabla.

Nombre	Tipo	Descripción
`Semaphore`	Clase	Proporciona el mecanismo clásico para regular el acceso de los a recursos de uso limitado.
`CountDownLatch`	Clase	Proporciona una ayuda para la sincronización cuando los hilos deben esperar hasta que se realicen un conjunto de operaciones.
`CyclingBarrier`	Clase	Proporciona una ayuda para la sincronización cuando unos hilos deben esperar hasta que otros hilos alcancen un punto de ejecución determinado.
`Phaser`	Clase	Proporcionauna funcionalidad similar a `CountDownLatch` y a `CyclingBarrier` a través de un uso más flexible.
`Exchanger`	Clase	Permite intercambiar elementos entre dos hilos.

Consulta el Ejemplo09

2.5. Estructuras de datos concurrentes

Java dispone de interfaces y clases para almacenar información con prácticamente cualquier tipo de estructura de datos. Interfaces heredadas de la interface `Collection`, como `List`, `Map`, `Set`, `Qeue` o `Deque`, son la base de una serie de clases que, implementando dichas interfaces, proporcionan el soporte necesario para todo tipo de estructuras de datos.

Desde el punto de vista de la concurrencia, estas implementaciones no están diseñadas para soportar que múltiples hilos lean y escriban simultáneamente en ellas, pudiendo provocar errores.

En el Ejemplo10, varios hilos leen y escriben en un objeto `ArrayList` compartido.

La ejecución de este código provoca múltiples errores de concurrencia.

La clase `Collections` de Java proporciona métodos estáticos para convertir estructuras de datos en seguras respecto a hilos (thread-safe), como `synchronizedList`, `synchronizedMap` o `synchronizedSet` entre otros, pero no son las alternativas más eficientes en todos los casos.

Como alternativa más eficiente si la mayor parte de las operaciones son de lectura, el paquete `java.util.concurrent` proporciona implementaciones específicamente diseñadas para entornos de ejecución multihilo. Algunas de esas clases se muestran en la siguiente tabla.

Nombre	Tipo	Descripción
`ConcurrentHashMap`	Clase	Equivalente a un `HashMap` sincronizado.
`ConcurrentSkipListMap`	Clase	Equivalente a un `TreeMap` sincronizado.
`CopyOnWriteArrayList`	Clase	Equivalente a un `ArrayList` sincronizado.
`CopyOnWriteArraySet`	Clase	Equivalente a un `Set` sincronizado.

El ejemplo anterior, referente a los lectores y escritores trabajando conjuntamente sobre un `ArrayList`, se convierte en thead-safe utilizando la clase `CopyOnWriteArraySet` en lugar de `ArrayList`, tal y como se puede observar en el Ejemplo11.

2.6. Las interfaces `ExecutorService`, `Callable` y `Future`

Usadas de manera conjunta, estas tres interfaces proporcionan mecanismos para ejecutar el código de manera asíncrona.

`ExecutorService` proporciona el marco de ejecución de código asíncrono contenido en objetos de las interfaces `Callable` y `Runnable`. Sus principales métodos se muestran en la siguiente tabla.

Método	Descripción
`awaitTermination`	Bloquea el servicio cuando se recibe una petición de apagado hasta que todas las tareas que tenía asignadas han terminado o se ha alcanzado el tiempo límite E de espera (timeout) o ha habido una interrupción.
`invokeAll`	Permite lanzar una colección de tareas y recoger una lista de objetos `Future`.
`invokeAny`	Permite lanzar una colección de tareas y recoger el resultado de la que termine satisfactoriamente (si alguna lo hace).
`shutdown()`	Hace que que están el servicio en ejecución deje de y finaliza. aceptar nuevas tareas, espera a que terminen
`shutdownNow()`	Finaliza el servicio sin esperar a que las tareas que tiene en ejecución lo hagan.
`submit()`	Envía a ejecución una tarea `Runnable` o `Callable`.

La construcción de instancias de `ExecutorService` se realiza a través de una serie de métodos estáticos de la clase Executors. Estos métodos permiten indicar la estrategia de hilos que desea que siga el ExecutorService. Algunos de los métodos estáticos de Executors para crear objetos de la interfaz ExecutorService son:

- `Executors.newCachedThreadPool()`: Crea un `ExecutorService` con un pool de hilos con todos los que sean necesarios, reutilizando aquellos que están libres.
- `Executors.newFixedThreadPool(int nThreads)`: Crea un `ExecutorService` con un pool con un número determinado de hilos.
- `Executors.newSingleThreadExecutor()`: Crea un `ExecutorService` con un único hilo.

Por su parte, la interfaz `Callable` es una interfaz que funciona similar a `Runnable`, pero con la diferencia de que puede devolver un retorno y lanzar una excepción. Es una interfaz funcional, ya que solo tiene el método call, por lo que se puede utilizar en expresiones lambda. El código asíncrono de un objeto `Callable` se ejecuta a través del método submit de `ExecutorService`.

La interfaz `Future` representa un resultado futuro generado por un proceso asíncrono. De alguna manera, es un sistema que permite dejar en suspenso la obtención de un resultado hasta que este está disponible. El resultado de invocar al método `submit` de un `ExecutorService` es un objeto `Future`.

Los métodos de la interfaz `Future` se recogen en la siguiente tabla.

Método	Descripción
`cancel`	Intenta cancelar la ejecución de la tarea.
`get`	Espera a que para termine la tarea y obtiene el resultado. En una de sus formas admite un timeout para limitar el tiempo de espera.
`isCancelled()`	Indica si la tarea se ha cancelado antes de terminar.
`isDone()`	Indica si la tarea ha terminado.

En resumen, la relación entre estas tres interfaces es la siguiente: `ExecutorService` ejecuta un objeto `Callable` (o `Runnable`) con una estrategia multihilo determinada y deposita en un objeto `Future` el retorno futuro de la tarea.

Consulta el Ejemplo12

> En programación, casi nunca una alternativa es mejor que otra siempre. Las interfaces `Runnable` y `Callable` son absolutamente válidas y, dependiendo de la situación, una u otra serán la mejor alternativa.

3. Ejemplos

3.1. Ejemplo01

Ejemplo con un único hilo de ejecución:

```
package UD02.Ejemplo01;

public class Raton {

    private String nombre;
    private int tiempoAlimentacion;

    public Raton(String nombre, int tiempoAlimentacion) {
        super();
        this.nombre = nombre;
        this.tiempoAlimentacion = tiempoAlimentacion;
    }

    public void comer() {
        try {
            System.out.printf("El raton %s ha comenzado a alimentarse%n", nombre);
            Thread.sleep(tiempoAlimentacion * 1000);
            System.out.printf("El ratón %s ha terminado de alimentarse%n", nombre);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Raton fievel = new Raton("Fievel", 4);
        Raton jerry = new Raton("Jerry", 5);
        Raton pinky = new Raton("Pinky", 3);
        Raton mickey = new Raton("Mickey", 6);
        fievel.comer();
        jerry.comer();
        pinky.comer();
        mickey.comer();
    }
}
```

La salida producida por la ejecución es la siguiente:

```
El raton Fievel ha comenzado a alimentarse
El ratón Fievel ha terminado de alimentarse
El raton Jerry ha comenzado a alimentarse
El ratón Jerry ha terminado de alimentarse
El raton Pinky ha comenzado a alimentarse
El ratón Pinky ha terminado de alimentarse
El raton Mickey ha comenzado a alimentarse
El ratón Mickey ha terminado de alimentarse
```

3.2. Ejemplo02

Ejemplo multihilo:

```java
package UD02.Ejemplo02;

public class Raton extends Thread {

    private String nombre;
    private int tiempoAlimentacion;

    public Raton(String nombre, int tiempoAlimentacion) {
        super();
        this.nombre = nombre;
        this.tiempoAlimentacion = tiempoAlimentacion;
    }

    public void comer() {
        try {
            System.out.printf("El raton %s ha comenzado a alimentarse%n", nombre);
            Thread.sleep(tiempoAlimentacion * 1000);
            System.out.printf("El ratón %s ha terminado de alimentarse%n", nombre);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        this.comer();
    }

    public static void main(String[] args) {
        Raton fievel = new Raton("Fievel", 4);
        Raton jerry = new Raton("Jerry", 5);
        Raton pinky = new Raton("Pinky", 3);
        Raton mickey = new Raton("Mickey", 6);
        fievel.start();
        jerry.start();
        pinky.start();
        mickey.start();
    }
}
```

Ejemplo de salida de su ejecución:

```
El raton Fievel ha comenzado a alimentarse
El raton Mickey ha comenzado a alimentarse
El raton Pinky ha comenzado a alimentarse
El raton Jerry ha comenzado a alimentarse
El ratón Pinky ha terminado de alimentarse
El ratón Fievel ha terminado de alimentarse
El ratón Jerry ha terminado de alimentarse
El ratón Mickey ha terminado de alimentarse
```

Todos los ratones han comenzado a alimentarse de inmediato, sin esperar a que termine ninguno de los demás. El tiempo total del proceso será, aproximadamente, el tiempo del proceso más lento (en este caso, 6 segundos). La reducción del tiempo total de ejecución es evidente.

3.3. Ejemplo03

Retomando el enunciado del ejemplo de los objetos de la clase Raton, la solución mediante implementación de la interface Runnable tendría el código que se muestra a continuación, revisa las partes del código que se han modificado respecto de la solución anterior en la que se heredaba de Thread.

```
1  package UD02.Ejemplo03;
2
3  public class Raton implements Runnable {
4
5      private String nombre;
6      private int tiempoAlimentacion;
7
8      public Raton(String nombre, int tiempoAlimentacion) {
9          super();
10         this.nombre = nombre;
11         this.tiempoAlimentacion = tiempoAlimentacion;
12     }
13
14     public void comer() {
15         try {
16             System.out.printf("El raton %s ha comenzado a alimentarse%n", nombre);
17             Thread.sleep(tiempoAlimentacion * 1000);
18             System.out.printf("El ratón %s ha terminado de alimentarse%n", nombre);
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22     }
23
24     @Override
25     public void run() {
26         this.comer();
27     }
28
29     public static void main(String[] args) {
30         Raton fievel = new Raton("Fievel", 4);
31         Raton jerry = new Raton("Jerry", 5);
32         Raton pinky = new Raton("Pinky", 3);
33         Raton mickey = new Raton("Mickey", 6);
34         new Thread(fievel).start();
35         new Thread(jerry).start();
36         new Thread(pinky).start();
37         new Thread(mickey).start();
38     }
39 }
```

El resultado de ejecución:

```
El raton Fievel ha comenzado a alimentarse
El raton Mickey ha comenzado a alimentarse
El raton Pinky ha comenzado a alimentarse
El raton Jerry ha comenzado a alimentarse
El ratón Pinky ha terminado de alimentarse
El ratón Fievel ha terminado de alimentarse
El ratón Jerry ha terminado de alimentarse
El ratón Mickey ha terminado de alimentarse
```

Se puede observar que el código es muy similar, salvo en la declaración de la clase (implementación de una interface frente a herencia) y en la creación de los hilos y su arranque: los objetos `Runnable` deben «envolverse» en objetos `Thread` para poder ser arrancados. Es importante apreciar que en ambos casos la ejecución se realiza invocando al método `start`.

3.4. Ejemplo04

Ejemplo multihilo a partir de un único.

```java
/*
 * Copyright (C) 2022 David Martínez (wwww.martinezpenya.es|www.ieseduardoprimo.es)
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */
package UD02.Ejemplo04;

/**
 *
 * @author David Martínez (wwww.martinezpenya.es|www.ieseduardoprimo.es)
 */
public class RatonSimple implements Runnable {

    private String nombre;
    private int tiempoAlimentacion;
    private int alimentoConsumido;

    public RatonSimple(String nombre, int tiempoAlimentacion) {
        super();
        this.nombre = nombre;
        this.tiempoAlimentacion = tiempoAlimentacion;
    }

    public void comer() {
        try {
```

```
37              System.out.printf("El raton %s ha comenzado a alimentarse%n", nombre);
38              Thread.sleep(tiempoAlimentacion * 1000);
39              alimentoConsumido++;
40              System.out.printf("El ratón %s ha terminado de alimentarse%n", nombre);
41              System.out.printf("Alimento consumido: %d%n", alimentoConsumido);
42          } catch (InterruptedException e) {
43              e.printStackTrace();
44          }
45      }
46
47      @Override
48      public void run() {
49          this.comer();
50      }
51
52      public static void main(String[] args) {
53          RatonSimple fievel = new RatonSimple("Fievel", 4);
54          new Thread(fievel).start();
55          new Thread(fievel).start();
56          new Thread(fievel).start();
57          new Thread(fievel).start();
58      }
59  }
60
```

El resultado de la ejecución es el siguiente:

```
1  El raton Fievel ha comenzado a alimentarse
2  El raton Fievel ha comenzado a alimentarse
3  El raton Fievel ha comenzado a alimentarse
4  El raton Fievel ha comenzado a alimentarse
5  El ratón Fievel ha terminado de alimentarse
6  Alimento consumido: 1
7  El ratón Fievel ha terminado de alimentarse
8  El ratón Fievel ha terminado de alimentarse
9  Alimento consumido: 4
10 El ratón Fievel ha terminado de alimentarse
11 Alimento consumido: 4
12 Alimento consumido: 4
```

Cada hilo ha ejecutado el método `run` sobre los datos del mismo objeto. Es decir, se ha ejecutado simultáneamente cuatro veces un bloque de código de un único objeto, compartiendo sus atributos. De esta forma, en la salida se puede apreciar que el valor del atributo `alimentoConsumido` se ha incrementado en 1 por cada hilo. Esto se puede observar porque el valor de `alimentoConsumido` se ha incrementado varias veces durante la ejecución. El hecho de que algunos valores intermedios no aparezcan en la siguiente captura de la salida de la ejecución tiene que ver con la `asincronía` y el alto coste de ejecución que tienen las sentencias que escriben por pantalla.

Aunque se tratará más adelante en esta misma unidad, es importante insistir en que los diferentes hilos del ejemplo anterior están trabajando sobre una única copia del objeto en memoria, por lo que las variables (los atributos) son compartidas, pudiendo sufrir errores de concurrencia.

3.5. Ejemplo04bis

Por ejemplo, sustituyendo el método main del ejemplo anterior por el siguiente código, se pueden crear y ejecutar varios threads mediante un bucle for a partir de una única instancia de una clase que implementa la interfaz Runnable. El resultado, con un número bajo de iteraciones (por ejemplo, 4) será habitualmente correcto (el atributo alimentoConsumido alcanzará el mismo valor que el número de iteraciones). Con un número alto (por ejemplo, 1000 iteraciones) el resultado será habitualmente erróneo (el atributo alimentoConsumido alcanzará un valor por debajo del número de iteraciones). Esto se debe a que al compartir todos los hilos los mismos atributos producen errores de concurrencia que deben evitarse mediante técnicas concretas que veremos más adelante.

```
public static void main(String[] args) {
    RatonSimple fievel = new RatonSimple("Fievel", 4);
    for (int i = 0; i < 100; i++) {
        new Thread(fievel).start();
    }
}
```

En una ejecución del código anterior, el resultado es 98, cuando debería haber sido 100 si no hubiese habido problemas de concurrencia

```
Alimento consumido: 98
```

3.6. Ejemplo05

Listado de estados por los que pasa un hilo:

```
    public static void main(String[] args) {
        RatonSimple mickey = new RatonSimple("Mickey", 6);
        ArrayList<Thread.State> estadosHilo = new ArrayList();
        Thread h = new Thread(mickey);
        estadosHilo.add(h.getState());
        h.start();

        while (h.getState() != Thread.State.TERMINATED) {
            if (!estadosHilo.contains(h.getState())) {
                estadosHilo.add(h.getState());
            }
        }
        if (!estadosHilo.contains(h.getState())) {
            estadosHilo.add(h.getState());

        }
        for (Thread.State estado : estadosHilo) {
            System.out.println(estado);
        }
    }
```

Al finalizar la ejecución se muestran los estados recogidos:

```
El raton Mickey ha comenzado a alimentarse
El ratón Mickey ha terminado de alimentarse
Alimento consumido: 1
NEW
RUNNABLE
TIMED_WAITING
TERMINATED
```

3.7. Ejemplo06

En el siguiente ejemplo se muestra un uso conjunto de las clases `Timer` y `TimerTask`. El programa simula controlar un sistema de regadío automático. Este sistema riega por primera vez transcurridos `1000` milisegundos desde el inicio de la ejecución y repite el riego cada `2000` milisegundos.

```java
import java.util.Timer;
import java.util.TimerTask;

public class SistemaRiego extends TimerTask {

    @Override
    public void run() {
        System.out.println("Regando...");
    }

    public static void main(String[] args) {
        Timer temporizador = new Timer();
        temporizador.schedule(new SistemaRiego(),1000,2000);
    }
}
```

La salida generada transcurridos unos segundos es la que se muestra a continuación. Entre la escritura de cada línea transcurren 2000 milisegundos.

```
Regando...
Regando...
Regando...
...
```

3.8. Ejemplo07

El siguiente ejemplo ilustra el uso de `ScheduledExecutorService` como herramienta para la programación de ejecuciones de tareas. Como se puede observar, mediante la clase `Executors` se obtiene una instancia de `ScheduledExecutorService`, que es la interface que permite programar tareas recurrentes en hilos independientes. Una vez obtenido el programador de tareas, se le indica qué tarea se quiere ejecutar (objeto `sr`), cuántas unidades de tiempo se desea esperar hasta que se inicie la primera tarea (1), cuántas unidades de tiempo se desea esperar entre cada repetición de la tarea (2) y en qué unidad están representadas las unidades de tiempo (`TimeUnit.SECONDS`)

```java
import java.util.concurrent.Executors;
```

```java
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class SistemaRiego implements Runnable {

    @Override
    public void run() {
        System.out.println("Regando...");
    }

    public static void main(String[] args) {
        SistemaRiego sr = new SistemaRiego();
        ScheduledExecutorService stp = Executors.newSingleThreadScheduledExecutor();
        stp.scheduleAtFixedRate(sr, 1, 2, TimeUnit.SECONDS);
        System.out.println("Sistema de riego configurado");
    }
}
```

La salida de la ejecución transcurridos unos segundos es la que se muestra a continuación. Entre cada escritura del texto «Regando» transcurren dos segundos.

```
Sistema de riego configurado
Regando...
Regando...
Regando...
```

3.9. Ejemplo08

Para ilustrar mejor el comportamiento de este tipo de soluciones se presenta el siguiente ejemplo. utiliza Una cola recibe escrituras y lecturas de un conjunto de hilos. La primera solución utiliza una estructura de datos `LinkedList` como un soporte. Al no ser esta estructura segura frente a múltiples hilos, la ejecución produce un error.

```java
import java.util.LinkedList;
import java.util.Queue;

public class ColaNoConcurrente implements Runnable {

    private static Queue<Integer> cola = new LinkedList<Integer>();

    @Override
    public void run() {
        cola.add(10);
        for (Integer i : cola) {
            System.out.print(1 + ":");
        }
        System.out.println("Tamaño cola:" + cola.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(new ColaNoConcurrente()).start();
        }
    }
```

```
}
```

La salida será similar a esta:

```
Exception in thread "Thread-2" Exception in thread "Thread-1" Exception in thread
"Thread-0" java.util.ConcurrentModificationException
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Tamaño cola:4
Tamaño cola:8
Tamaño cola:9
Tamaño cola:7
1:1:1:1:1:1:1:1:1:Tamaño cola:10
    at
java.base/java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:970)
[...]
```

3.10. Ejemplo08bis

La segunda solución utiliza el mismo código cambiando la estructura de datos a `ConcurrentLinkedDeque` obteniendo una ejecución sin errores.

```
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedDeque;

public class ColaConcurrente implements Runnable {

    private static Queue<Integer> cola = new ConcurrentLinkedDeque<Integer>();

    @Override
    public void run() {
        cola.add(10);
        for (Integer i : cola) {
            System.out.print(1 + ":");
        }
        System.out.println("Tamaño cola:" + cola.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(new ColaConcurrente()).start();
        }
    }
}
```

Salida obtenida:

```
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Tamaño cola:4
Tamaño cola:4
Tamaño cola:7
Tamaño cola:8
Tamaño cola:4
Tamaño cola:4
1:1:1:1:1:1:1:1:1:1:Tamaño cola:10
1:Tamaño cola:5
1:1:1:1:1:1:1:1:1:Tamaño cola:10
Tamaño cola:6
```

Aunque de esta última ejecución no está ordenada debido a la concurrencia de los hilos, sí se puede observar que la ejecución ha generado una cola con 10 elementos correspondientes a los 10 hilos que estaban accediendo a leer y escribir en la estructura de datos.

3.11. Ejemplo09

En el siguiente ejemplo se muestra el uso de `Exchanger`. Se crean dos clases que implementan `Runnable`, `TareaA` y `TareaB`. Ambas clases reciben una instancia de `Exchanger` en el constructor y la utilizan para intercambiar información entre sí. La llamada al método `exchange` por parte de una de las dos tareas producirá un bloqueo de espera hasta que la otra tarea haga lo propio, intercambiando en ese momento la información entre los dos hilos. Por su parte, la clase `Intercambiador`, construye tanto el objeto `Exchanger` como las dos tareas programadas en `TareaA` y `TareaB`. Es importante prestar atención al hecho de que las tareas no tienen referencias la una de la otra, sino que tienen acceso al objeto que hace de «intercambiador» de información.

`TareaA`:

```java
import java.util.concurrent.Exchanger;

public class TareaA implements Runnable {

    private Exchanger<String> exchanger;

    public TareaA(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            String mensajeRecibido = exchanger.exchange("Mensaje enviado por TareaA");
            System.out.println("Mensaje recibido en TareaA: " + mensajeRecibido);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

`TareaB`:

```java
import java.util.concurrent.Exchanger;

public class TareaB implements Runnable {

    private Exchanger<String> exchanger;

    public TareaB(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            String mensajeRecibido = exchanger.exchange("Mensaje enviado por TareaB");
            System.out.println("Mensaje recibido en TareaB: " + mensajeRecibido);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

`Intercambiador`:

```java
import java.util.concurrent.Exchanger;

public class Intercambiador {

    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<String>();
        new Thread(new TareaA(exchanger)).start();
        new Thread(new TareaB(exchanger)).start();
    }
}
```

La salida obtenida será:

```
Mensaje recibido en TareaA: Mensaje enviado por TareaB
Mensaje recibido en TareaB: Mensaje enviado por TareaA
```

3.12. Ejemplo10

```java
package UD02.Ejemplo10;

import java.util.ArrayList;
import java.util.List;

public class LectorEscritorNoSeguro extends Thread {

    private static List<String> palabras = new ArrayList<String>();

    @Override
```

```
    public void run() {
        palabras.add("Nueva palabra");
        for (String palabra : palabras) {
            palabras.size();
        }
        System.out.println(palabras.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new LectorEscritorNoSeguro().start();
        }
    }
}
```

En la salida aparecen referencias a excepciones que indican que han ocurrido errores de concurrencia en el acceso a la lista, como `java.util.ConcurrentModificationException`:

```
[...]
Exception in thread "Thread-21" java.util.ConcurrentModificationException
[...]
```

3.13. Ejemplo11

```
package UD02.Ejemplo11;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class LectorEscritorSeguro extends Thread {

    private static List<String> palabras = new CopyOnWriteArrayList<String>();

    @Override
    public void run() {
        palabras.add("Nueva palabra");
        for (String palabra : palabras) {
            palabras.size();
        }
        System.out.println(palabras.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new LectorEscritorSeguro().start();
        }
    }
}
```

Que esta vez se ejecuta sin problemas ni errores.

3.14. Ejemplo12

```java
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Lector implements Callable<String> {

    @Override
    public String call() throws Exception {
        String textoLeido = "Me gustan las películas de acción";
        Thread.sleep(1000);
        return textoLeido;
    }

    public static void main(String[] args) {
        try {
            Lector lector = new Lector();
            ExecutorService servicioEjecucion = Executors.newSingleThreadExecutor();
            Future<String> resultado = servicioEjecucion.submit(lector);
            String texto = resultado.get();
            if (resultado.isDone()) {
                System.out.println(texto);
                System.out.println("Proceso finalizado");
            } else if (resultado.isCancelled()) {
                System.out.println("Proceso cancelado");
            }
            servicioEjecucion.shutdown();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

La salida producida es la siguiente:

```
Me gustan las películas de acción
Proceso finalizado
```

Como se ha indicado anteriormente, la principal diferencia entre implementar `Callable` y `Runnable` es que la primera opción puede proporcionar un retorno. No obstante, con `Runnable` existen técnicas para transmitir valores mediante el uso de referencias a objetos.

4. Fuentes de información

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos